Scalable Collision Detection Using p-Partition Fronts on Many-Core Processors

Xinyu Zhang, *Member*, *IEEE*, and Young J. Kim, *Member*, *IEEE*

Abstract—We present a new parallel algorithm for collision detection using many-core computing platforms of CPUs or GPUs. Based on the notion of a p-partition front, our algorithm is able to evenly partition and distribute the workload of BVH traversal among multiple processing cores without the need for dynamic balancing, while minimizing the memory overhead inherent to the state-of-the-art parallel collision detection algorithms. We demonstrate the scalability of our algorithm on different benchmarking scenarios with and without using temporal coherence, including dynamic simulation of rigid bodies, cloth simulation, and random collision courses. In these experiments, we observe nearly linear performance improvement in terms of the number of processing cores on the CPUs and GPUs.

Index Terms—Collision detection, p-partition, static workload balancing

1 Introduction

Recent advances in parallel processors such as multicore CPUs and many-core GPUs have made parallel computing ubiquitous, and such trends are expected to continue in the future. However, to fully benefit from the development of modern computing hardware, many existing collision detection algorithms such as [1], [2], [3], [4], designed mainly for a single processor, have to be redesigned for parallel processors, due to the high data dependence in the sequential algorithms and data access latency from the memory architecture.

Recently, a few algorithms have been proposed for parallel collision detection [5], [6], [7], [8], [9], [10], [11]. Most of them use sophisticated dynamic workload balancing algorithms, for instance, work stealing, to provide the scalability of collision detection in terms of the number of processing cores. Even though the dynamic balancing mechanisms employed by these approaches show some promising results, their successful implementation heavily depends on the underlying hardware architecture such as memory hierarchy or caching, and processor communication channel [12], since the centralized/decentralized scheduler utilizes these architectures to synchronize the global coordination among processors. As a result, different collision detection algorithms have to be redesigned depending on the underlying computing platform (e.g., CPUs, GPUs, their hybrid, or distributed processors). Moreover, on some platform such as GPUs, it is known that dynamic balancing mechanism for collision detection is very inefficient due to the lack of sophisticated caching mechanism and data latency among computing cores [13].

On the other hand, *static workload balancing* can eliminate such synchronization and communication overhead. However, in some sense, static balancing can be even more challenging to devise than dynamic ones since the workload must be determined prior to execution.

Meanwhile, bounding volume hierarchies (BVHs) are the most popular data structure adopted by many researchers to accelerate collision queries [14]. For a pair of geometric objects, BVHs are used to efficiently localize the potentially colliding substructure of geometries by recursively bounding each object with a set of bounding volumes (BVs). Notably, Tang et al. [9] used the bounding volume test tree (BVTT) front and its decomposition for workload distribution among multiple processing cores. However, maintaining such a BVTT front can lead to a significant overhead on memory footprint. Since the size of a BVTT front grows rapidly with respect to the model complexity, especially when the models are in close proximity, the system memory can quickly run out even for medium-size models; for instance, a small model consisting of 4K triangles requires about 17M system memory ($O(m^2)$ memory complexity) [9] and a model consisting of 1.6M triangles requires over 3G system memory [10] to maintain its BVTT front. Moreover, computing a BVTT in a parallel-friendly fashion is not obvious and also subject to load balancing.

Thus, despite the exciting recent progress in the field of parallel collision detection, designing a scalable algorithm with low-memory footprint, applicable to both multicore CPUs or many-core GPUs, remains a challenging problem.

Main results. In this paper, we present a new scalable collision detection algorithm for many-core computing platforms (CPUs and GPUs). Our algorithm utilizes a novel workload partitioning representation of BVTT, called a p-partition front, which enables static and even partitioning of the workload of parallel BVH traversals, and thus, complicated dynamic load balancing can be avoided at

- *X. Zhang is with the Department of Computer Science and Engineering, Ewha Womans University, Seoul, South Korea, and also with the Software Engineering Institute, East China Normal University, China. E-mail: zhangxy@ewha.ac.kr.*
- *Y.J. Kim is with the Department of Computer Science and Engineering, Ewha Womans University, Seoul, South Korea. E-mail: kimy@ewha.ac.kr.*

Manuscript received 4 Sept. 2012; revised 14 Feb. 2013; accepted 20 Sept. 2013; published online 3 Oct. 2013.
Recommended for acceptance by J. Keyser.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2012-09-0181.
Digital Object Identifier no. 10.1109/TVCG.2013.239.

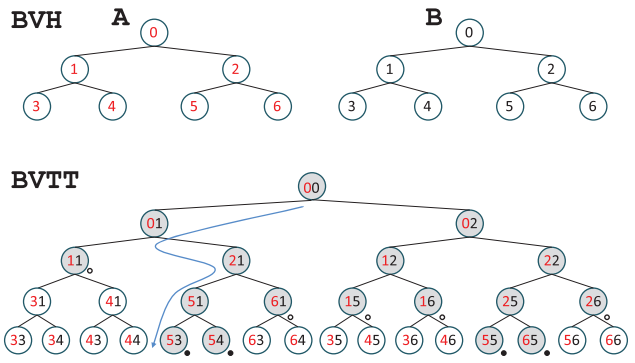


Fig. 1. BVH and BVTT. Top: BVHs of two objects A and B. Bottom: BVTT is denoted by the gray nodes representing a snapshot of BVH traversal. One traversal path is highlighted in blue. The nonoverlapping, internal BVH nodes are marked by a circle at bottom-right corner and the nodes testing between two primitive nodes are marked by a black solid dot. The maximal front of a BVTT consists of all these marked nodes.

runtime. Because of this, our algorithm is amenable to an efficient and scalable implementation of parallel collision detection on different computing platforms. Moreover, our algorithm requires a small memory footprint compared to other existing BVH-based algorithms. In our experiments, our algorithm shows nearly linear scalability in terms of computing cores and is able to achieve up to 11 times performance speedup using two Intel hexacore CPUs and 30-45 times speedups using NVIDIA GPUs, compared to the CPU-based, sequential algorithm.

2 Preliminaries

In this section, we provide some preliminary concepts to describe our algorithm.

2.1 BVH and BVTT

In BVH-based collision detection, before invoking the expensive primitive-level intersection tests between objects, tests based on simple volumes (e.g., AABB) that bound the objects are executed to prune away redundant primitive-level tests. These algorithms perform the BVH traversal either in depth first, breadth first, or hybrid manner. The BVH traversal starts with a pair of root BVH nodes and is recurrently applied to their child nodes. If the recursive traversal reaches leaf nodes, the two corresponding primitives are tested for intersection; otherwise, the recursion continues for their child nodes.

A BVTT represents a snapshot of BVH traversal [15]. Each node in the BVTT corresponds to a single-overlap test between a pair of BV primitives or between a pair of triangle primitives. The root node of a BVTT is a BV test between the roots of BVHs. The size of a BVTT means the number of nodes contained in the BVTT. Fig. 1 illustrates two BVHs and their BVTT.

The front of a BVTT is a boundary of the BVTT at runtime. The maximal front of a BVTT refers to the leaves of the BVTT, corresponding to either a test between two BVH leaf nodes or between a pair of nonoverlapping, internal BVH nodes. Typically, the number of the nodes of a maximal front is about a half of that of the entire BVTT. An intermediate front of BVTT appears at an intermediate state of a BVTT during BVH traversal. An intermediate front of BVTT consists of a set of BVTT internal nodes, which evolves from the BVTT root and gradually reaches the BVTT maximal front. Each of these nodes in the intermediate front is the ancestor of a node of the maximal front.

2.2 Notation

Let $\mathcal{F}$ denote a front of a BVTT. Then, let $\mathcal{F}_I$ be an intermediate front of a BVTT, and $\mathcal{F}_M$ be the maximal front of a BVTT. Let n be a node of a BVTT. We use $|n|$ to denote the cost of traversing the BVH substructure corresponding to the node n, which can be approximated by the number of BV-level primitive tests and triangle-level primitive tests that need to be executed. Since both of the primitive tests can be regarded as a constant, if a triangle test is k times expensive than a BV test, then we can denote $|n| = (m_1 + k \cdot m_2)$, where m_1 and m_2 are the number of a BV test and the number of a triangle test, respectively. We denote $|\mathcal{F}| = \sum |n|$ for all $n \in \mathcal{F}$.

2.3 Static versus Dynamic Workload Balancing

Regarding workload balancing on GPUs and many-core systems, it is generally believed that static workload balancing is preferred when the process workloads, hardware speed, network bandwidth, and application behavior are known a priori and they do not change over time or change only slightly. In this case, static workload balancing is superior to dynamic one due to the implementation simplicity and minimal runtime synchronization overhead.

Dynamic workload balancing becomes effective when the workloads among processors and application behavior tend to shift dramatically during the lifetime of computation. Runtime monitoring and redistribution of workload can improve the efficiency of workload balancing mechanism. However, the benefits of using dynamic workload balancing can be reduced by the overheads associated with monitoring/detecting workloads, workload redistribution, and potential undesirable balancing disturbances ("oscillation" phenomenon) induced by the rebalancing algorithm itself. Such overheads can vary widely with different hardware and workload states. Finally, it is quite nontrivial to implement a good and efficient dynamic workload balancing algorithm.

Regarding performing parallel BVH traversal for collision detection, it is tempting to use dynamic load balancing because 1) the collision workload is not known a priori; 2) the initial workload is very low and has one task to execute at the root node; and 3) the workload of each node in BVTT is typically unbalanced. Furthermore, successful implementations of dynamic load balancing heavily depend on the underlying hardware architecture such as memory hierarchy, caching, and processor communication channel. On some many-core platform such as GPUs, it is known that dynamic balancing mechanism (particularly for collision detection) is very inefficient due to the lack of sophisticated caching mechanism and/or data latency among computing cores [13]. As analyzed by Lauterbach et al. [16], none of the known dynamic load balancing algorithms have been successfully immigrated to many-core GPU architectures. Please refer to [13], [16] for more details.

3 Static Workload Partition

Using the maximal front of a BVTT, a perfect workload balancing can be achieved for parallel BVH traversals since the workload can be fully identified, as well as fine grained. However, this requires keeping track of the maximal front, which can incur huge memory overhead [7], [17]. Moreover, when objects move or deform, the corresponding maximal front changes, and thus needs to be updated accordingly [15].

As a matter of fact, as long as effective load balancing can be obtained from some intermediate front of a BVTT, we do not need the maximal front for parallel BVH traversals, not to mention the high computational and memory overhead to maintain the maximal front. Thus, we introduce a new type of a front, called a p-partition front that can greatly simplify the workload distribution during BVH traversals, while having little overhead on memory footprint. Intuitively speaking, a p-partition front corresponds to any front whose workload can be evenly distributed to p parallel processors.

3.1 The p-Partition Front

We first give the definition of p-partition, which our algorithm builds upon. Given a finite set Π of positive integers, its partition is a finite collection of subsets $\pi = (\pi_1, \pi_2, \ldots, \pi_p)$, where $\pi_1, \pi_2, \ldots, \pi_p$ are pairwise disjoint and nonempty sets whose union is Π. p is the size of π and $\pi_1, \pi_2, \ldots, \pi_p$ are the partitions of π. If the element sum of each partition $|\pi_i|$ in π is equal for all partitions, we call π a perfect p-partition of Π and Π is a p-partition set. However, a perfect p-partition may not exist and, even if it does exist, computing it may be computationally infeasible [18]. Therefore, it is more practical to find the *best* p-partition. Finding the best p-partition is a problem of optimization: given a set Π and an integer p, find a partition π so to minimize $\max_\pi(|\pi_1|, |\pi_2|, \ldots, |\pi_p|)$, where $|\pi_i|$ is the sum of elements in π_i $(i = 1, 2, \ldots, p)$. However, even the problem of finding the best p-partition is NP-hard and the solutions to the problem are generally derived using heuristics and approximation [19]. Moreover, the hardness of finding the best p-partition increases as the size of Π decreases or the element value increases or p increases [20].

Now, we define the p-partition front for parallel BVTT traversals.

p-Partition front. Given a front $\mathcal{F}$ of a BVTT and p processors, if the nodes of $\mathcal{F}$ can be separated into p disjoint subsets $(\ldots, \mathcal{F}_i, \ldots)$ such that $\mathcal{F} = \cup\mathcal{F}_i$ and $|\mathcal{F}_i| = \frac{|\mathcal{F}|}{p}$ $(i = 1, \ldots, p)$, $\mathcal{F}$ is said to be a (perfect) p-partition front and $(\ldots, \mathcal{F}_i, \ldots)$ is a (perfect) p-partition of $\mathcal{F}$. We denote a p-partition front by $\mathcal{F}_p$.

For a BVTT, there may exist many p-partition fronts. We also define:

Minimal p-partition front. Is the p-partition front which has the minimum number of nodes. We denote the minimal p-partition front by $\mathcal{F}_p^{min}$.

Maximal p-partition front. Is the p-partition front which has the maximal number of nodes. We denote the maximal p-partition front by $\mathcal{F}_p^{max}$.

As shown in Fig. 2, the green and black solid curves illustrate the minimal and the maximal p-partition fronts, respectively. Between these two p-partition fronts, there exist other intermediate p-partition fronts (dotted curve). These intermediate p-partition fronts can be generated by dropping the nodes from the minimal p-partition front or raising the nodes from the maximal p-partition front. Typically, the size of the minimal p-partition front is significantly smaller than that of the maximal p-partition front. The maximal front of BVTT is equivalent to the maximal p-partition front and the partition of the maximal front is a partition of the finest grained front.

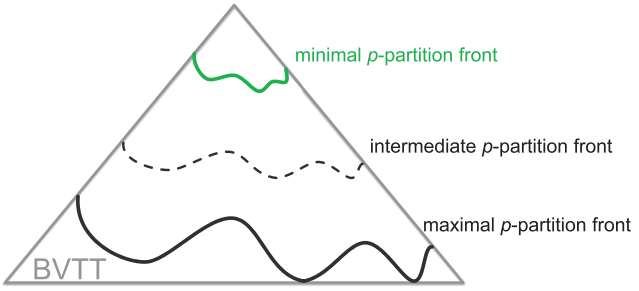


Fig. 2. A BVTT and A p-Partition Front. The minimal p-partition front has the minimum number of nodes and the maximal p-partition front has the maximal number of nodes.

Our goal is to find the front of a minimal size (i.e., the minimal p-partition) whose workload can be evenly distributed to p parallel processors. This can avoid generating a large intermediate p-partition front, especially avoiding generating the maximal p-partition front (i.e., the maximal front of a BVTT). Although a large intermediate p-partition front or the maximal p-partition front has many nodes which can be partitioned in such a way to have very evenly balanced workloads, its memory and computational cost preclude realizing this theoretical benefit in practice. An ideal approach to generating the minimal p-partition front is first described in Section 3.2. Then, we propose a more practical solution to approximate the minimal p-partition front by utilizing temporal coherence (see Section 3.2.1) and without using temporal coherence (see Section 3.2.2).

3.2 Generating a p-Partition Front

To generate the minimal p-partition front, one may start BVH traversal from the root of BVTT and advancing the front of BVTT gradually to the minimal p-partition front. During each step of BVH traversal, we add a node n to front $\mathcal{F}$ and check whether $\mathcal{F}$ is a p-partition front (be p-partitioned) based on the costs of its nodes. If $\mathcal{F}$ is not a p-partition front, we continue the BVH traversal by visiting the children of n. The BVH traversal terminates when $\mathcal{F}$ is found to be a p-partition front. The pseudocode of this algorithm is given in Algorithm 1. However, this algorithm assumes that the cost of any node n is known, which, however, cannot be precisely determined until actually finishing the BVH traversal. Moreover, performing p-partition checking at each step of traversal slows down the algorithm considerably. Therefore, we propose an algorithm to approximate these values in the following sections. And we propose two sets of algorithms to approximate the minimal p-partition front, by utilizing temporal coherence and without relying on such coherence.

Algorithm 1. `traverse`(BVH a, BVH b).
a, b [input]: two BVH nodes
$\mathcal{F}$ [output]: the minimal p-partition front

```
if (a ∩ b == ∅) then
    return
end if
push [n(a,b), |n(a,b)|] onto F
check whether F is a p-partition front
if F is a p-partition front then
    terminate
    return F as the minimal p-partition front
else
    for all children a_i and b_j do
        traverse(a_i, b_j)
    end for
end if
```

3.2.1 Using Temporal Coherence

Many applications such as physically based animation or sampling-based motion planning exhibit temporal coherence between successive time steps. In this case, the costs of BVH traversals from the previous frame can be used for estimating the costs for the next frame. However, maintaining the costs for all the nodes in a BVTT or even part of a BVTT is still memory intensive. Moreover, as discussed before, frequent testing of p-partition during the BVH traversal makes the algorithm inefficient. Therefore, we present a new algorithm to approximate the minimal p-partition front. The algorithm includes three steps: 1) initializing the starting BVTT front that stores the cost of a BVH traversal based on temporal coherence; 2) generating a front of a BVTT in which the costs of all the nodes are bounded; and 3) finding the best p-partitions from the resulting front of a BVTT. We elaborate this algorithm as follows:

Algorithm 2. `p_partition_front`(BVH a, BVH b).
$\mathcal{F}_\mathrm{s}$ [input]: a starting front cached as preprocessing
$\mathcal{F}_\mathrm{p}$ [output]: an approximation of the minimal p-partition front

```
for each node n in F_s do
    if |n_s^{t-1}| < |F_s^{t-1}| / (g·p) then
        push [n_s, |n_s^{t-1}|] onto F_p
    else
        push [n_s, |n_s^{t-1}|] onto Q
        while Q is not empty do
            pop a node [n, |n|] off Q
            for each n's child n_ij = (a_i, b_j) do
                if overlap(a_i, b_j) then
                    calculate the overlap volume V_ij
                else
                    V_ij = 0
                end if
            end for

            for each n's overlap child n_ij do
                |n_ij| = V_ij / (Σ V_ij) · (|n| − 1)

                if |n_ij| < |F_s^{t-1}| / (g·p) then
                    push [n_ij, |n_ij|] onto F_p
                end if
            end for
        end while
    end if
end for
return F_p as an approximation of the minimal
    p-partition front
```

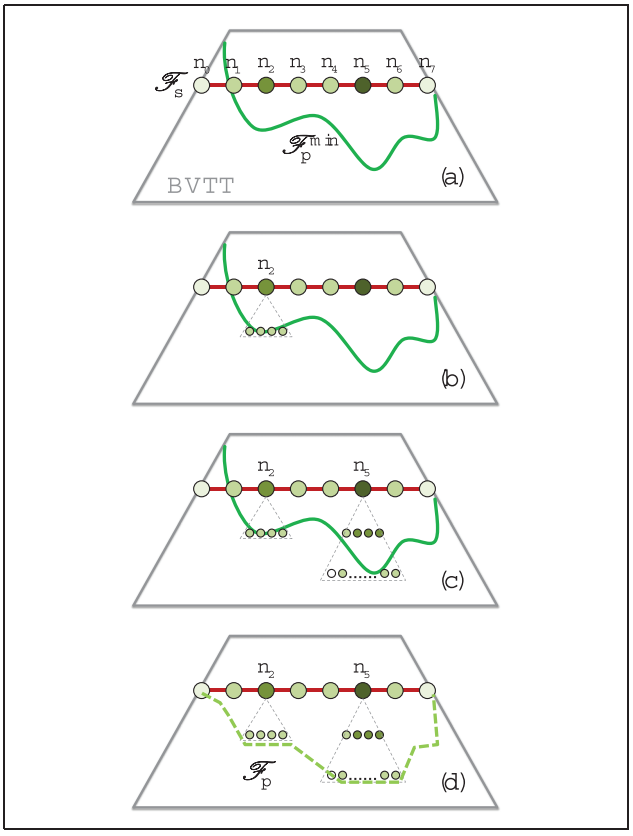


Fig. 3. Approximating the minimal p-partition front. (a) From a starting front $\mathcal{F}_s$, we approximate the minimal p-partition front $\mathcal{F}_p^{min}$ (green solid curve) by examining each node of $\mathcal{F}_s$ using (1). (b) Since $|n_2| > \frac{|\mathcal{F}_s^{t-1}|}{g \cdot p}$, it needs to be traversed until its children satisfy (1). (c) $|n_5|$ is much greater than $|n_2|$, so n_5 needs more traversals than n_2. (d) A p-partition front is obtained after finishing the execution.

First of all, rather than starting the BVH traversal from its root node, we predetermine a starting front $\mathcal{F}_s$ as preprocessing. Then, at runtime, for each node $n_s \in \mathcal{F}_s$, we save its cost $|n_s^{t-1}|$ at time frame $t-1$. The total cost of $\mathcal{F}_s$ is $|\mathcal{F}_s^{t-1}| = \sum |n_s^{t-1}|$. As illustrated in Fig. 3, the nodes of starting front $\mathcal{F}_s$ are shown as the green shading circles. Here, a darker circle indicates a higher cost node; for example, $|n_5| > |n_2| > |n_0|$.

Then, we examine each $n_s \in \mathcal{F}_s$ by checking whether its cost is less than the average cost of p-partition front, that is

$$|n^{t-1}| < \frac{|\mathcal{F}_s^{t-1}|}{g \cdot p}. \quad (1)$$

Here, a user-specified threshold g is used to adjust the workload grains in the resulting p-partition front. In our implementation, we choose $g = 2$.

If a node satisfies (1), we immediately add it to the p-partition front ($\mathcal{F}_p$); otherwise, we expand it by visiting its

children. As shown in Fig. 3, both n_2 and n_5 do not satisfy (1) and need to traverse the BVTT further until their children satisfy (1). While traversing n_2 and n_5, we need to know the children's costs to examine whether any of them will satisfy (1). And if any of them satisfies (1), its cost will be eventually used to determine the best p-partition of the front.

Based on our experiments and also the approach suggested by Klein and Zachmann [21] for AABBs, the costs of the four children nodes (i.e., the number of BV tests and triangle tests) are linear to the relative size of their overlap volumes. Note that the precise number of BV tests and triangle tests may be very hard to estimate merely using the size of their overlap volume [21], [9] for all the levels of a BVTT. However, since we know the cost of a node, the costs of its children nodes can be more accurately estimated using the sizes of their overlap volumes. If we let the cost function be $|n_{ij}| = \alpha_{ij} \cdot V_{ij}$, where V_{ij} is the size of the overlap volume of n, and α_{ij} is a function for capturing the other geometric aspect of the cost such as geometric shape or relative configurations [22], the cost of a child node $n_{ij} = n(a_i, b_j)$ can be formulated as

$$|n_{ij}| = \frac{\alpha_{ij} \cdot V_{ij}}{\sum \alpha_{ij} \cdot V_{ij}} \cdot (|n| - 1), (i, j = 1, 2). \quad (2)$$

If we assume that $\alpha_{ij} = \alpha$(constant), then we obtain

$$|n_{ij}| = \frac{V_{ij}}{\sum V_{ij}} \cdot (|n| - 1), (i, j = 1, 2). \quad (3)$$

Alternatively, the size of an overlap volume can be replaced by a penetration depth [8] to estimate $|n_{ij}|$, which can be used for OBB.

In addition, for each node $n \in \mathcal{F}_p$, we store the backward index to its associated starting node from which the node n sprouts. For example, all the p-partition front nodes generated from n_5 have an index 5 to indicate that they sprouted from n_5. Later, this index is used to update the costs of $\mathcal{F}_s$, which can be used for the next time frame. We execute these expansions and evaluations of generating a p-partition front in parallel.

3.2.2 Using Overlap Volume Only

When motion coherence is not present, we use another approach to estimate the cost of BVH traversal. Though there are some algorithms [21] to estimate the expected cost of BVH traversal, predicting the exact cost can be very difficult and inaccurate. Moreover, we do not need absolute costs, but relative ones serve our purpose. As we discussed earlier, the cost of BVH traversal largely depends on the overlap volume. Similarly, we let cost function be $|n_i| = \alpha_i \cdot V_i$, where V_i is the size of overlap volume of node $n_i \in \mathcal{F}_s$. If we assume that the function α_i remains constant for the given two objects, then we have the following condition:

$$V_i < \frac{\sum V_i}{g \cdot p}, \quad (4)$$

to determine whether a node is to be added to the p-partition front.

In the case of nontemporal coherence, we first compute the overlap volume for all nodes in $\mathcal{F}_s$ in parallel. Then, we use (3) and (4) to expand the nodes and approximate the p-partition front. Note that the overlapping volume heuristic is used both in our temporal and nontemporal coherence solutions.

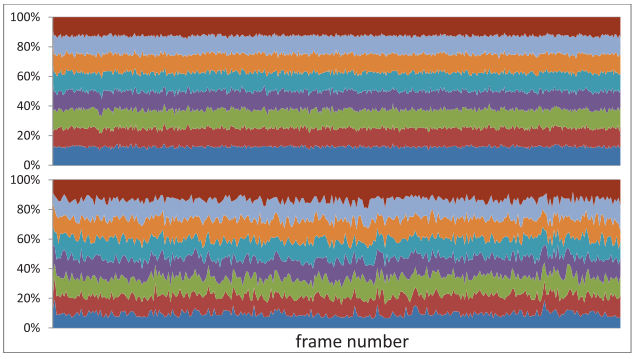


Fig. 4. Workload distribution among eight cores for the worm-gear simulation and different colors denote different workloads. Top: percentages of triangle primitive tests for each thread. Bottom: percentages of BV tests for each thread.

3.3 Executing p-Partitioning

After obtaining a p-partition front, we need to determine the best p-partition of the front and each partition of the front will be assigned to a separate processing core. However, since determining the best p-partitioning is an NP-hard problem, we use a greedy algorithm to approximate it. More specifically, we first sort the p-partition front with respect to the node cost in descending order. Then, as we go through all these nodes, we take p largest nodes and assign each of them to the partition subset. For the rest of nodes, we add them successively to which set is the smallest. This greedy algorithm gives a $\frac{p+2}{p+1}$ approximation [23].

3.4 Parallel BVH Traversal

In case of the simulation with temporal coherence, at the beginning (the first frame) of simulation, we assume that all the nodes of the starting front $\mathcal{F}_s$ have the same cost, and thus initialize $|n_s| = 1$ ($n_s \in \mathcal{F}_s$). Considering that the size of $\mathcal{F}_s$ is much greater than the number of processing cores p, this assumption easily makes $\mathcal{F}_s$ be a p-partition front without further BVH traversal. Then, we evenly divide $\mathcal{F}_s$ into p-partitions. Each processing core independently processes one of these partitions. Note that, in general, the workload will not be well balanced for the first frame due to this assumption.

During the parallel BVH traversal, we record the costs of BVH traversal and save them back to the starting front $\mathcal{F}_s$, which will be used for the next frame. After the first frame, the workload will be well balanced. In Fig. 4, the workload distribution among eight processing cores is shown for a sequence of worm-gear simulation (refer to Fig. 6a for the simulation). The percentage of workload (the number of BV tests and triangle tests) for each core is displayed in different colors. The figure also shows the evolution of workload percentages that each computing core consumes over the course of simulation. This figure nicely illustrates that the workload has been evenly distributed among eight processing cores using our algorithm.

For the case without using temporal coherence, it is not necessary to record the history of BV tests and triangles tests. Once a p-partition front is generated, it can be readily

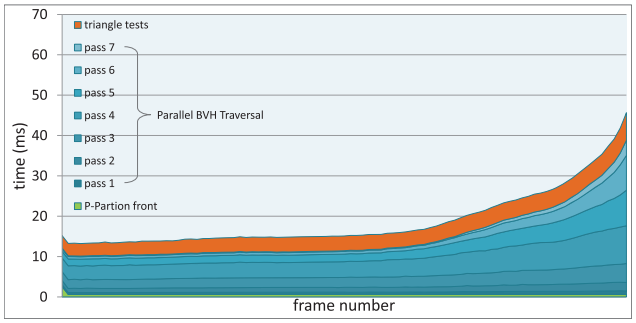


Fig. 5. Cloth Simulation. The times spent on depth-bounded p-partition front generation (green), parallel BVH traversal (gradient blue), and triangle overlap tests (orange). The individual pass of parallel BVH traversal is displayed in gradient blue.

divided into p-partitions and assigned to individual processing cores. Our experiments show that the scenarios with temporal coherence tend to have better workload balancing than those without temporal coherence.

Our approach is also applicable to the scenarios that consist of multiple objects. In this case, we individually maintain a starting front for each pair of objects. To generate a global p-partition front for all object pairs, we examine whether the cost of each node from all the starting fronts or their children is less than some global upper bound as follows:

$$|n^{t-1}| < \frac{\sum |\mathcal{F}_s^{t-1}|}{g \cdot p}. \quad (5)$$

Fig. 6b shows an example of multiple objects. This implies that our algorithm is also applicable to the scenarios with topological changes, for instance, when an object breaks into multiple pieces.

4 The Depth-Bounded p-Partition Front on GPUs

Compared to multicore CPUs, many-core GPUs have more processing cores and are able to launch significantly more concurrent threads for parallel BVH traversals. As we discussed earlier, finding the best p-partition front gets harder as the number of cores p increases, which makes it very challenging for many-core GPUs to generate a p-partition front.

Here, we propose a new algorithm to address this issue. The main idea of our algorithm for GPUs is based on the fact that finding a best p-partition gets easier as the size of the target set increases or the cost of nodes decreases; thus, we make a larger starting front and constrain the cost of a node by limiting its depth of BVH traversal. The latter constraint makes the high-cost nodes terminate their BVH traversal earlier when reaching the depth bound. Another benefit is that the workload will be fine grained. For example, if we limit the depth bound to 2, the cost of any node during BVH traversal will be less than $4^2 = 16$ (BV tests). For the nodes that have not completed traversing the BVH even after reaching the depth bound, we leave them to the next stage. More precisely, our depth-bounded algorithm generally takes a few passes to complete the entire BVH traversal; a smaller depth bound incurs more passes of BVH traversal. Moreover, since a triangle-level primitive test is generally more expensive than the BV-level primitive test, it is likely to introduce unbalanced workload on GPUs if the BV-level and triangle-level primitive tests are performed all together. Thus, during the BVH traversal, if a leaf node corresponding to a pair of triangles is found, we save it and defer the triangle overlap test to the final step.

In addition, unlike the case of multicore CPUs, we do not actually partition the resulting p-partition front, because its time complexity can be as high as $O(p^2 N_p^2)$ using the greedy algorithm [23], where N is the size of p-partition front; note that on GPUs, both p and N can be large. Instead, we utilize the *transparent scalability* offered by the modern GPU computing architecture such as CUDA [24] to partition and distribute the workload. In detail, we simply collect all the nodes from the p-partition front, uniformly map each of them to thread blocks, and let CUDA automatically decide how many threads should be allocated to finish the job. Our experiment shows that this approach is very efficient for HW-supported workload balancing since the workload is fine grained.

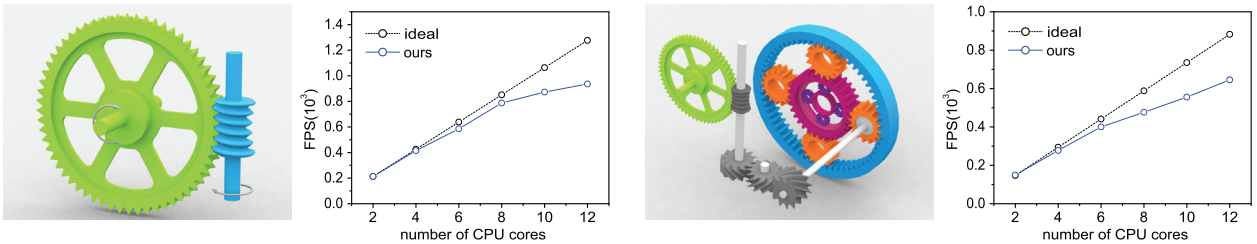
An example is given in Fig. 5 to show the time spent on each stage of our GPU-based collision detection algorithm, including depth-bounded p-partition front generation, parallel BVH traversal, and triangle-level overlap tests displayed in different colors. The parallel BVH traversal generally needs a few passes to complete. The series of gradient color (blue) are used to differentiate these passes.

5 Results and Analysis

We will now explain some implementation details of our algorithms and show our experimental results.

5.1 Implementation

We have implemented our parallel collision detection algorithm using C++ and CUDA under Visual Studio 2008 and Windows 7. We used the public-domain collision



(a) Worm Gears (b) Epicyclic-Gears

Fig. 6. Dynamic simulation using temporal coherence. (a) A worm-gear system consisting of 64K triangles. (b) A epicyclic gears system consisting of 17 gears and 149K triangles in total.

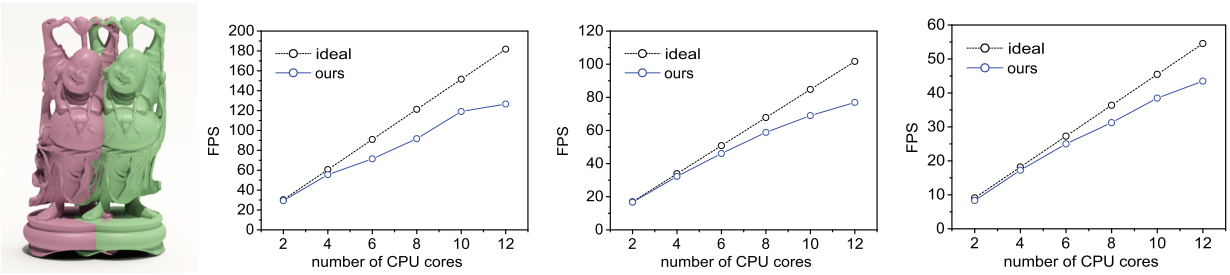


Fig. 7. A sequence of motion for two Buddha models. Each Buddha is composed of more than 500K, 1M, and 3M triangles, respectively.

detection software, OPCode [25] for rigid bodies and used DeformCD [26] for cloth simulation. The CPU algorithms were implemented with OpenMP and the GPU algorithm was implemented with NVIDIA CUDA 4.2. The algorithms were tested on a workstation with two 3.47-GHz Intel Xeon X5690 hexacore CPUs and NVIDIA Tesla C2070/GeForce GTX 680 GPUs.

The BVH traversal and triangle-level overlap tests require random access to GPU global memory, which may affect the performance. To handle this issue, in our algorithm, we utilize GPU textures to maintain BVHs and triangle vertices and indices, as GPU texture memory offers efficient addressing and fetching modes and also provides memory caching capability.

5.2 Experimental Results

5.2.1 Dynamic Simulation

We applied our CPU-based parallel algorithm to the mechanical dynamic simulation using temporal coherence. The worm-gear benchmark (see Fig. 6a) consists of 64K triangles and the epicyclic gears system (see Fig. 6b) consists of 17 gears and 149K triangles in total. Our algorithm is able to achieve 7.5 times speedup for eight cores. However, when the number of cores is greater than eight, the performance does not improve significantly though the CPU utilization is still observed as 100 percent. The main reason is because the OpenMP overhead becomes non-negligible when the collision detection time falls 1 ms per frame as shown in Fig. 6. Fig. 6b also demonstrates that our algorithm is applicable to the scenario consisting of many objects.

5.2.2 Collision Sequence between Complex Models

We generated a sequence of motions for a pair of complex models (the Buddha models). The model complexity varies from 500K to 3M triangles. During the motion, the red Buddha penetrates the green one from left to right. We applied our temporal coherence solution to this experiment. As shown in Fig. 7, the result shows a near-linear performance improvement.

5.2.3 Cloth Simulation

We tested our algorithms on UNC Cloth Benchmarks[1] using temporal coherence, and measured its performance on multicore CPUs and many-core GPUs. The cloth consists of 92K triangles and self-collision detection is performed. We have observed nearly linear performance improvement with respect to the number of CPU cores (see Fig. 8), and 30 and 45 times performance improvement NVIDIA Tesla C2070 GPUs and GeForce GTX 680 GPUs over the sequential CPU version.

5.2.4 Random Collision Courses

In this experiment, we randomly generated collision configurations for two bunny models (see Fig. 9), consisting of 58K triangles each. We applied our nontemporal coherence solution to this experiment. As shown in Fig. 9, the result shows nearly linear performance improvement before the time falls 1 ms per frame.

5.3 Discussions

Generating a p-partition front is performed in a parallel manner and the time spent on it is typically less than 1 percent in our experiments.

We use a threshold g for controlling the size of workload grains in the p-partition front. In all benchmarks, we choose $g = 2$. With an increment of g, the size of workload grains can be reduced, which can increase the possibility of having a good partition. However, a high g may cause overhead because deep BVH traversals are required.

In our implementation, the size of a starting front varies from 4^3 to 4^5 for the CPU algorithm. For the GPU algorithm, the starting front ranges from 4^6 to 4^7. Generally, the size of a starting front can be determined based on the model complexity and the overlap volume of the BVTT root. For a

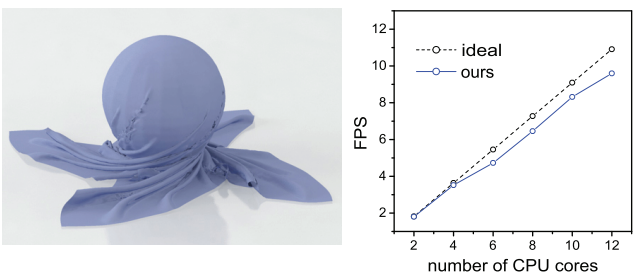


Fig. 8. Cloth simulation: performance speedup with respect to the number of CPU cores.

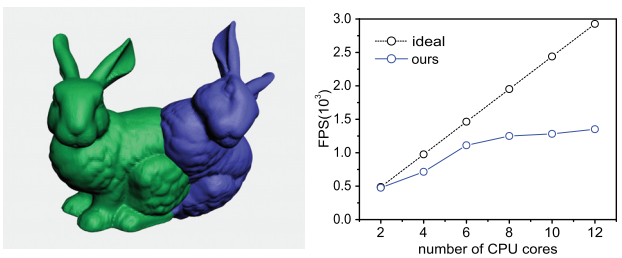


Fig. 9. Random collision courses for two Bunny models. Update rates (FPS) with respect to the number of CPU cores.

1. http://gamma.cs.unc.edu/DYNAMICB.

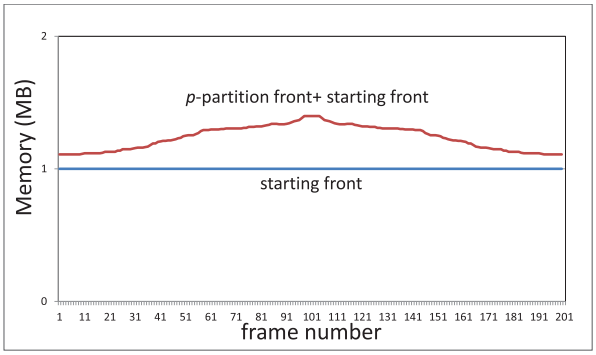


Fig. 10. The memory usage profile of p-partition front $O(N_s + N_p)$ for the sequence of motion of two Buddha models (when $m = 500$K and $p = 4$).

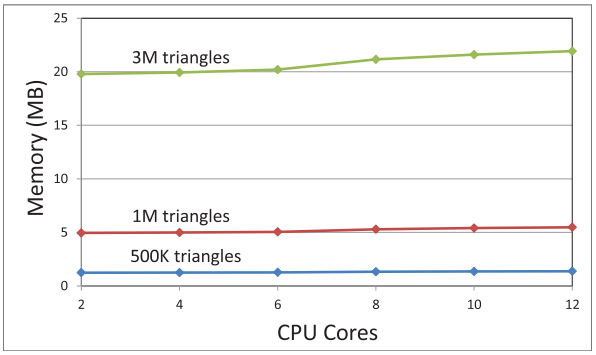


Fig. 11. The average memory usage of p-partition front: $O(N_s + N_p)$ w.r.t the number of CPU cores and w.r.t the model complexity for the benchmark of two Buddha models whose complexities vary from 500K to 3M triangles.

less (more) complex model, we need a smaller (larger) starting front. When the overlap volume changes at runtime, we slightly adjust the size of the starting front.

The performance scalability of our algorithm is sensitive to the pattern of BVH traversal, but is less sensitive to model complexity (except simple models). In general, a parallel algorithm tends to exhibit a higher scalability as the problem size increases [27]. In our case, it is more likely to find an optimal p-partition front if the benchmarks consist of complex models and exhibit complex collision scenarios. In contrast, if the problem size is relatively small, it is not easy to devise a scalable algorithm [27]. If the BVTT lacks parallelism, for instance, the BVTT is skewed, our approach is not very scalable in this case. However, our approach can still help to identify the optimal number of cores for the efficient parallel BVH traversal. For instance, whenever a node n was found to have a much higher cost than others after a few iterations of traversals, it can be considered as a dominating path. In this case, the optimal number of processing cores can be estimated as

$$\mathrm{p}_{optimal} = \left\lceil \frac{|\mathcal{F}_s^{t-1}|}{|n^{t-1}|} \right\rceil. \tag{6}$$

The space complexity of our algorithm is $O(N_s + N_p + p \cdot (\log m - d_s))$ in the worst case, where N_s is the size of the starting front and N_p is the size of the resulting p-partition front, m is the number of triangles, $\log m$ is the depth of the BVH, and d_s is the depth of starting front $\mathcal{F}_s$. For a given pair of models, $O(N_s + N_p)$ corresponds to generating a p-partition front and $O(p(\log m - d_s))$ is for the runtime parallel BVH traversal on multicore CPUs. For a pair of models, N_s is a constant (4^{d_s}) and N_p can be bounded by $N_p < N_s$ ($g = 2$). Note that the nonoverlapping nodes are not included in the resulting p-partition front. The depth-bounded p-partition front on GPUs requires $O(N_s + N_p + p \cdot d_b)$ space, where d_b is the depth bound (e.g., 2). It is clear that the space requirement can be reduced by choosing a smaller depth bound (d_b), which corresponds to a higher number of passes to complete the parallel BVH traversal. Fig. 10 shows the memory usage profile (i.e., $O(N_s + N_p)$) to generate the p-partition fronts in the benchmark of two Buddha models for a certain number of CPU cores ($p = 4$). Fig. 11 shows the average memory usage of generating the p-partition fronts (i.e., $O(N_s + N_p)$) on multicore CPUs for the same benchmark of two Buddha models with complexities from 500K to 3M triangles. In summary, generating a p-partition front shows a linear growth of memory usage in terms of model complexities. For a given number of processor cores (p), the space complexity of runtime BVH traversal is logarithmic with respect to the model complexity and the corresponding space scalability with respect to p is linear in the worst case.

5.4 Comparisons

In this section, we make some qualitative comparisons of our algorithm against the state-of-the-art collision works in terms of the scalability of parallelism on CPUs and GPUs. First of all, it is very nontrivial to quantitatively compare the performance of our method over prior parallel collision detection methods, mainly because different algorithms utilize different computing platforms and some of them are far outdated or poorly scalable in modern parallel processors [27]. Thus, in this section, we merely compared the performance of our algorithms against what were reported in the original work.

For CPU-based algorithms, [6] can achieve seven times speedup using eight-core CPUs (two Intel 2.83-GHz quad-core CPUs and OpenMP) using dynamic balancing on the cloth benchmark, which is a little bit worse than ours. Moreover, ours show further scalability for even a higher number of cores. In other dynamic balancing work, Tang et al. [9] show 6.7 and 10.5 times performance improvement using 8 and 16 cores (two Intel 2.93-GHz octacore, 16GKB memory, and OpenMP) based on BVTT front tracking. Combined with deferred front tracking [10], the performance can be slightly improved (20 percent improvement in their experiments), but its memory footprint is still significantly high compared to ours (NVIDIA GeForce GTX 480 and CUDA). Lee and Kim [8] show up to five times improvement using eight cores, both of which are worse than ours. Other prior CPU-based parallel algorithms such as [28] (12 MIPS1000 processors, 250 MHz, 4GBK memory) and [29] (unknown hardware) suffered from poor parallel scalability. In summary, without relying on sophisticated dynamic workload balancing mechanism, our parallel algorithm based on static workload balancing is able to achieve the scalability comparable to or better than dynamic ones (e.g., work stealing) in our benchmarks. Thus, our algorithm is useful for parallel computing platforms with or without communication/data delays among cores or sophisticated memory hierarchy. Also, note

that some of the earlier algorithms such as [6] and [9] use continuous collision detection (CCD) unlike our discrete collision detection algorithm; however, in this case, our p-partitioning could be more effective since the BVTT front size for CCD is generally larger than that for discrete collision detection.

On the GPU side, the algorithm in [7] (CUDA 4.0, NVIDIA GTX285 GPUs) achieves 7-12 times performance improvement using a central workload distribution scheme, compared to the state-of-the-art CPU algorithm, whereas our algorithm shows 30-45 times improvement on the same benchmark on slightly better GPUs.

6 Conclusion

We have introduced a new approach to parallel collision detection using multicore CPUs and many-core GPUs. Our approach simplifies the workload balancing for the BVH traversal using p-partition front, which is statically determined before parallel BVH traversal is performed. The computational overhead for approximating the minimal p-partition front turned out to be negligible. The good performance of our algorithm is attributed to the static and even workload partitioning that can be locally processed by individual processors. Our workload balancing scheme considerably eliminates the need for synchronization and communication overhead. Reducing the memory footprint size has a significantly positive impact on the performance and scalability of our algorithm as the model complexity increases. Thus, our algorithm was able to achieve nearly linear performance speedup with respect to the number of computing cores.

There are a few limitations in our algorithm. The scalability of our algorithm is output sensitive. Thus, if the collision detection result requires a small amount of computations (e.g., simple model and/or shallow overlap), our algorithm becomes less efficient. In case that the BVTT lacks in obvious parallelism, dynamic workload balancing such as work stealing is a better option than our technique. Our algorithm to find the best p-partition front is greedy such that there is no guarantee to find an optimal one. For a very high number of CPU cores (i.e., high p), the time spent on finding the best p-partition will be non-negligible, which may affect the performance of our algorithm. Our algorithm requires a few problem-dependent parameters such as g and α.

For future work, we would like to further investigate the extensions of our algorithm to other proximity queries such as distance computation and penetration depth. In addition, our algorithm is able to benefit dynamic workload balancing algorithms such as work stealing since the even workload distribution at the initial stage can significantly reduce the runtime communication and synchronization overhead between processing cores for dynamic balancing. We would like to extend our collision framework work to heterogeneous computing platforms such as hybrid CPUs and GPUs. We also would like to apply our techniques to continuous collision detection.

Acknowledgments

This work was supported in part by NRF in Korea (nos. 2012R1A2A2A01046246, 2012R1A2A2A06047007). Y.J. Kim is the corresponding author.

References

[1] S. Gottschalk, M.C. Lin, and D. Manocha, "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection," *Proc. ACM SIGGRAPH,* pp. 171-180, 1996.

[2] G. van den Bergen, "Efficient Collision Detection of Complex Deformable Models Using AABB Trees," *Graphics Tools,* vol. 2, no. 4, pp. 1-13, 1997.

[3] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, K. Zikan, "Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs," *IEEE Trans. Visualization and Computer Graphics,* vol. 4, no. 1, pp. 21-36, Jan. 1998.

[4] S.A. Ehmann and M.C. Lin, "Accurate and Fast Proximity Queries between Polyhedra Using Surface Decomposition," *Computer Graphics Forum,* vol. 20, no. 3, pp. 500-510, 2001.

[5] C.J. Hughes, R. Grzeszczuk, E. Sifakis, D. Kim, S. Kumar, A.P. Selle, J. Chhugani, M. Holliman, and Y. kuang Chen, "Physical Simulation for Animation Visual Effects: Parallelization Characterization for Chip MultiProcessors," *Proc. 34th Ann. Int'l Symp. Computer Architecture (ISCA '07),* 2007.

[6] D. Kim, J.-P. Heo, and S.-E. Yoon, "PCCD: Parallel Continuous Collision Detection," *Proc. ACM SIGGRAPH,* article 50, 2009.

[7] C. Lauterbach, Q. Mo, and D. Manocha, "gProximity: Hierarchical GPU-Based Operations for Collision and Distance Queries," *Computer Graphics Forum,* vol. 29, no. 2, pp. 419-428, 2010.

[8] Y. Lee and Y.J. Kim, "Simple and Parallel Proximity Algorithms for General Polygonal Models," *Computer Animation and Virtual Worlds,* vol. 21, pp. 365-374, 2010.

[9] M. Tang, D. Manocha, and R. Tong, "MCCD: Multi-Core Collision Detection between Deformable Models Using Front-Based Decomposition," *Graphical Models,* vol. 72, no. 2, pp. 7-23, 2010.

[10] M. Tang, D. Manocha, J. Lin, and R. Tong, "Collision-Streams: Fast GPU-Based Collision Detection for Deformable Models," *Proc. Symp. Interactive 3D Graphics and Games,* pp. 63-70, 2011.

[11] J. Pan and D. Manocha, "GPU-Based Parallel Collision Detection for Fast Motion Planning," *Int'l J. Robotics Research,* vol. 31, no. 2, pp. 187-200, 2012.

[12] D. Neill and A. Wierman, "On the Benefits of Work Stealing in Shared-Memory Multiprocessors," technical report, Carnegie Mellon Univ., 2009.

[13] D. Cederman and P. Tsigas, "Dynamic Load Balancing Using Work-Stealing," *GPU Computing Gems,* Elsevier, 2012.

[14] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum,* vol. 28, pp. 375-384, 2009.

[15] E. Larsen, S. Gottschalk, M.C. Lin, and D. Manocha, "Fast Proximity Queries with Swept Sphere Volumes," *Proc. Int'l Conf. Robotics and Automation,* pp. 3719-3726, 2000.

[16] C. Lauterbach, Q. Mo, and D. Manocha, "Work Distribution Methods on GPUs," technical report, Univ. of North Carolina at Chapel Hill, 2009.

[17] M. Tang, D. Manocha, and R. Tong, "Multi-Core Collision Detection between Deformable Models," *Proc. SIAM/ACM Joint Conf. Geometric and Physical Modeling,* pp. 355-360, 2009.

[18] S. Mertens, "Phase Transition in the Number Partitioning Problem," *Physical Rev. Letters,* vol. 81, pp. 4281-4284, 1998.

[19] R.E. Korf, "A Complete Anytime Algorithm for Number Partitioning," *Artificial Intelligence,* vol. 106, no. 2, pp. 181-203, 1998.

[20] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems.* Springer, 2004.

[21] J. Klein and G. Zachmann, "The Expected Running Time of Hierarchical Collision Detection," *Proc. ACM SIGGRAPH,* 2005.

[22] Y. Zhou and S. Suri, "Analysis of a Bounding Box Heuristic for Object Intersection," *Proc. ACM-SIAM Symp. Discrete Algorithms,* pp. 830-839, 1999.

[23] S. Mertens, "Computational Complexity and Statistical Physics," *The Easiest Hard Problem: Number Partitioning,* Oxford Univ. Press, 2006.

[24] NVIDIA, *NVIDIA CUDA C Programming Guide,* 2011.

[25] Pierre Terdiman, http://www.codercorner.com/opcode.htm, 2000.

[26] M. Tang and D. Manocha, "DeformCD: Collision Detection for Deformable Models [version 1.0]," http://gamma.cs.unc.edu/DEFORMCD, 2007.

[27] V. Kumar and A. Gupta, "Analyzing Scalability of Parallel Algorithms and Architectures," *J. Parallel and Distributed Computing,* vol. 22, no. 3, pp. 379-391, 1994.

[28] M. Figueiredo and T. Fernando, "An Efficient Parallel Collision Detection Algorithm for Virtual Prototype Environments," *Proc. Int'l Conf. Parallel and Distributed Systems,* 2004.
[29] I. Grinberg and Y. Wiseman, "Scalable Parallel Collision Detection Simulation," *Proc. Ninth IASTED Int'l Conf. Signal and Image Processing,* pp. 380-385, 2007.

Xinyu Zhang received the BS and MS degrees in material science, and the PhD degree in computer science from Zhejiang University in 1997, 2000, and 2004, respectively. He was a research professor from 2010 to 2012, a full-time lecturer from 2007 to 2008 and a postdoctoral research fellow from 2005 to 2007 in the Department of Computer Science and Engineering at Ewha Womans University. He is a researcher in the Department of Computer Science and Engineering at Ewha Womans University and a research scientist in the Department of Computer Science at the University of North Carolina at Chapel Hill. His research interests include computer graphics, geometric modeling, and collision detection. He is a member of the IEEE.

Young J. Kim received the PhD degree in computer science from Purdue University in 2000. He was a postdoctoral research fellow in the Department of Computer Science at the University of North Carolina at Chapel Hill. He is an associate professor of computer science and engineering at Ewha Womans University. His research interests include interactive computer graphics, computer games, robotics, haptics, and geometric modeling. He has published more than 50 papers in leading conferences and journals in these fields. He also received the Best Paper Awards at the ACM Solid Modeling Conference in 2003 and the International CAD Conference in 2008, and the best poster award at the Geometric Modeling and Processing conference in 2006. He was selected as best research faculty of Ewha in 2008, and received the Outstanding Research Cases Award from the Korean Research Foundation in 2008. He is a member of the IEEE.

Queries to the Author

Q1. Fig. 1 is not cited in the text. Please cite it at appropriate place.
Q2. We have used caligraphic font instead of mathscript font throughout the article. Please check.
Q3. Figure 2 has been changed to figure 3. Please verify for correctness.